

Übungsserie 10

Abgabe: gemäss Angaben Dozent

Scannen Sie ihre manuellen Lösungen für die Aufgaben 1 und 2 in die Dateien *Name_S10_Aufg1.pdf* bzw. *Name_S10_Aufg2.pdf* und fassen Sie diese mit Ihrer Python-Funktion *Name_S10_Aufg3a.py* und dem Skript *Name_S10_Aufg3b.py* in einer ZIP-Datei *Name_S10.zip* zusammen. Laden Sie dieses File vor der Übungsstunde nächste Woche auf Moodle hoch.

Aufgabe 1 (ca. 45 Minuten):

Gegeben ist das lineare Gleichungssystem

$$Ax = b \text{ mit } A = \begin{pmatrix} 8 & 5 & 2 \\ 5 & 9 & 1 \\ 4 & 2 & 7 \end{pmatrix} \text{ und } b = \begin{pmatrix} 19 \\ 5 \\ 34 \end{pmatrix}.$$

- Überprüfen Sie, ob das obige System bzgl. dem Jacobi-Verfahren konvergiert.
- Berechnen Sie auf vier Stellen nach dem Komma die Näherung $x^{(3)}$ mit dem Jacobi-Verfahren ausgehend vom Startvektor $x^{(0)} = \begin{pmatrix} 1 \\ -1 \\ 3 \end{pmatrix}$. Schreiben Sie alle benötigten Matrizen sowie die verwendete Iterationsgleichung explizit auf. Die Iterationen selber führen Sie aber natürlich mit Python durch.
- Wie gross ist gemäss der a-posteriori Abschätzung der absolute Fehler von $x^{(3)}$?
- Schätzen Sie a-priori die Anzahl Iterationsschritte ab, damit der berechnete Näherungsvektor in jeder Komponente maximal um 10^{-4} von der exakten Lösung $x = (2, -1, 4)^T$ abweicht.
- Wieviele Iterationsschritte würden Sie a-priori benötigen, wenn Sie als Startvektor nicht $x^{(0)}$ sondern $x^{(2)}$ aus b) verwenden würden?

Aufgabe 2 (ca. 30 Minuten):

Wiederholen Sie die obige Aufgabe, diesmal für das Gauss-Seidel Verfahren. Sie dürfen (ausnahmsweise) die Inverse von $D + L$ benutzen (müssen aber nicht, wenn Sie nicht wollen).

Aufgabe 3 (ca. 75 Minuten):

- Implementieren Sie das Jacobi- und Gauss-Seidel-Verfahren zusammen in einer Funktion als `[xn, n, n2] = Name_S10_Aufg3a(A,b,x0,to1,opt)`. Sie können dabei die Matrix-Funktionen von `numpy` und `numpy.linalg` in Python benutzen, (z.B. `triu(A)`, `diag(diag(A))`, `tril(A)`, `inv(D+L)`), ohne aber `inv(A)` zu berechnen. Dabei soll `xn` der Iterationsvektor nach `n` Iterationen sein, zusätzlich soll `n2` die Anzahl benötigter Schritte gemäss der a-priori Abschätzung angeben. Über den Parameter `opt` soll gesteuert werden, ob das Jacobi- oder das Gauss-Seidel Verfahren zur Anwendung kommt. Überlegen Sie sich, wie die Abbruchbedingung für Ihre `while`-Schleife lauten muss, um die Iteration bei Erreichen einer vorgegebenen Fehlertoleranz `to1` abzubrechen. Sie werden dafür

die Norm brauchen: `norm(..., np.inf)`. Achten sie darauf, dass Sie Matrizen, die Sie in ihrer Funktion nicht mehr brauchen, gleich wieder löschen, um Speicher freizugeben¹.

b) Schreiben Sie ein kurzes Skript `Name_S10_Aufg3b.m`. Testen Sie damit die Laufzeit Ihres Programmes für ihre Implementation des Jacobi- und Gauss-Seidel im Vergleich zum Gauss-Verfahren, welches Sie in Serie 6 implementiert hatten (siehe `Name_S6_Aufg2.m`) und im Vergleich zur Python-Funktion `np.linalg.solve()`. Verwenden Sie dafür die folgenden Werte für A, b, x_0 und tol :

```
>>> dim = 3000
>>> A = np.diag(np.diag(np.ones((dim, dim))*4000))+np.ones((dim, dim))
>>> dum1 = np.arange(1, np.int(dim/2+1), dtype=np.float64).reshape((np.int(dim/2), 1))
>>> dum2 = np.arange(np.int(dim/2), 0, -1, dtype=np.float64).reshape((np.int(dim/2), 1))
>>> x = np.append(dum1, dum2, axis=0)
>>> b = A*x
>>> x0 = np.zeros((dim, 1))
>>> tol = 1e-4
```

Den Zeitvergleich können Sie dabei analog wieder mit `timeit` messen (verzichten Sie auf 'repeat', da die Gauss-Zerlegung einige Zeit braucht ... lassen Sie die Gauss-Zerlegung deshalb nur laufen, wenn Sie Ihren Computer für einige Minuten nicht für anderes brauchen.).

Wieviel länger braucht Ihre eigene Gauss-Zerlegung als z.B. das Gauss-Seidel Verfahren? Schreiben Sie die gemessenen Werte als Kommentar in Ihr Programm.

c) Sie haben bei b) die "exakte" Lösung x definiert. Plotten Sie den absoluten Fehler für jedes Vektorelement ihrer drei Lösungsvektoren (d.h. Gauss, Jacobi, Gauss-Seidel). Was stellen Sie fest? Schreiben Sie Ihren Kommentar ins Skript.

¹In Python gibt es, im Gegensatz zu Matlab, keinen expliziten Befehl wie z.B. `clear D`, um alloziertes Memory für eine Matrix D zur Laufzeit wieder freizugeben. Der Befehl `del` in Python löscht lediglich die Verbindung zum Memoryspace, aber nicht die Variable selbst. Was hingegen funktioniert, ist eine nicht mehr gebrauchte Matrix zu überschreiben, z.B. mit `D = 0`, so dass der Garbage Collector das Memory wieder freigibt.